

EXHIBIT F

DECLARATION OF KELLY C. HUNSAKER IN SUPPORT OF MICROSOFT CORPORATION'S MOTIONS FOR SUMMARY JUDGMENT AND *DAUBERT* MOTIONS

IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF TEXAS
MARSHALL DIVISION

JUXTACOMM TECHNOLOGIES, INC.

PLAINTIFF,

v.

ASCENTIAL SOFTWARE CORPORATION,
et al.

DEFENDANTS.

§
§
§
§
§
§
§
§
§
§

Civil Action No. 2:07-CV-00359-LED

REBUTTAL EXPERT REPORT OF WALTER G. RUDD

13 separation between -- what's the word? -- OLE object
14 persistent repository, filters and transformations. No, it
15 does not refer to specific processor logic at this level of
16 a document. Hart depo. pp. 114-115.

Microsoft did conceive of the idea of a rowset before June 3, 1996. However, I have seen no documents that indicate that Microsoft had thought of data bags as claimed.

The materials I have reviewed do not disclose a pre-January 31, 1997 conception of either data bags, as construed by the Court, or of a rule set processor responsive to a script processor for manipulating a data bag for storing imported data and a data bag for storing export data. Dr. Carver claimed that Microsoft released Beta 1 of SQL Server on June 25, 1997, Beta 2 on December 16, 1997, Beta 3 on June 22, 1998, and finally released to market on November 16, 1998. Carver report § 45. It is my opinion that Beta 1 did not contain a systems interface as required by claim 2 of the Ellis-Doyle patent. It was not until Beta 2 on December 16, 1997 that a first version of a wizard appeared that could be construed as a systems interface as required by the claims. Certainly by Beta 3 Microsoft had a functioning version of DTS Designer and therefore a complete conception of a systems interface.

In summary, the earliest date at which Microsoft could have had the conception of the Ellis-Doyle inventions is December 16, 1997. There is no clear and convincing evidence that Microsoft had that conception before then, and it is my opinion that Microsoft did not have the conception of the Ellis-Doyle inventions before this date.

VII. THE WHITE REPORT

A. InfoPump

Mr. White claims that “InfoPump Version 1.0 (‘InfoPump’) anticipates and/or renders obvious claims 1-19 of the ‘662 patent.” I disagree. InfoPump is a software

Rebuttal Expert Report of
Walter G. Rudd

generator. It does not include scripts, a metadata database, a script processor, a rule set processor, or data bags.

1. InfoPump did not contain *1a) ... scripts*

InfoPump “scripts” are actually InfoPump Basic language programs that must be compiled to p-code instructions before being execution.

Another InfoPump Server component, the interpreter, is the component which performs the user-defined instructions for data movement, and manipulation. The term interpreter indicates the processing of p-code instructions from the compiler, but may actually be acting on a simple request which contains no p-code. An interpreter process is spawned for each request to be executed, and more than one of these processes may be active at a time. The interpreter process receives the request ID, execution ID, and InfoPump administrator password in its parameter list, and uses these to obtain Idb information.

In the case of scripted requests, the interpreter scans the compiled script header (see compiler description under InfoPump Manager), and allocates the data space required by the script. Then, the p-code statements are executed until the script ends or an unhandled error is encountered. In the case of simple requests, the code statements are bypassed since the instruction set is always the same. A function called RunQuery is streamlined to handle simple requests with maximum efficiency, bypassing unnecessary interpreter elements. In addition, information required for script generation (metadata and any clear command) is stored back in the request (see compiler description for the format of this information). During execution, information may be logged in the Idb in the Error log, Message log, and/or Created table and column logs. At the end of execution, any databases still connected are disconnected, execution information is written to the Idb (execution end time and modified table log data). DEFS00025509.

A script must be compiled before it can be tested or executed. If you compile a script and make changes to the script without recompiling, it's the original compiled version that's tested or executed. CA00177334.

Changing a code fragment requires recompiling the program. “Once you have compiled a script with included fragments, changing those fragments in InfoPump

Manager has no effect unless you recompile the script.” CA00177379. Similarly, changing a user-defined structure would require recompiling the program. Thus, a “script” in InfoPump is a static inter-communication facility as denigrated in the Ellis-Doyle patent.

The Ellis-Doyle patent distinguishes over systems like InfoPump in which programs are compiled and then executed. “To permit data exchange when different formats are used, a static inter-communication facility must be maintained for each pair of disparate data formats and/or data storage types. Changes to data formats or data storage types force the re-engineering of the corresponding facility.” Ellis-Doyle patent 1:24-30. Compiled scripted requests in InfoPump are static inter-communication facilities.

As Mr. White observes, the file history distinguished such systems.

Next, JuxtaComm argued that, “Morgenstern teaches a generation module 30 which outputs a compiled information mediator/bridge 60 adapted to convert data from a source format to data of a target format. Applicants acknowledge that the end result of the integration platform for heterogeneous databases taught by Morgenstern is equivalent to the end result achieved by the instant invention. Nonetheless, the methodology taught by Morgenstern teaches directly away from the claimed invention. Morgenstern’s compiled information mediator/bridge 60 is inflexible and adapted for single instance applications. If either the source or the target database structures are modified, the entire process must be re-executed and a new information mediator/bridge 60 compiled.” White report § 71.

As noted above, in InfoPump, as in Morgenstern, if either the source or the target database structures are modified, the entire process must be re-executed. InfoPump suffered from the same deficiencies as Morgenstern: it was inflexible and adapted for single instance applications. Thus Morgenstern teaches away from the Ellis-Doyle patent, as does InfoPump.

I conclude that InfoPump did not include the claimed scripts. It would not have been obvious to add scripts because InfoPump utilizes a compiled p-code approach that processes low-level p-code instructions at runtime for runtime efficiency. The InfoPump scripting language “is implemented as a machine-independent p-code compiler for execution speed and portability to other environments.” DEFS00097225. Given the concern for execution speed, it would not have been obvious to add scripts because runtime script processing would be slower in execution speed than p-code execution. Furthermore, adding scripts to InfoPump would have required a significant redesign and programming effort.

2. InfoPump did not include *1b) a metadata database for storing said logical import and export data interfaces, data transformation rule sets and scripts;*

InfoPump did not include the claimed scripts and so did not include a metadata database for storing scripts.

3. InfoPump did not include *1c) a script processor for utilizing metadata from the metadata database to control data transformation within said systems interface and movement of said data into and out of said distribution system*

Mr. White claims that “the InfoPump Server includes a software component that processes scripted requests...This software component is called the interpreter, and it utilizes metadata, such as the scripts, import and export data interfaces, and data transformation rule sets...stored in the InfoPump Database (the metadata database) to control data transformation within the systems interface and movement of data into and out of InfoPump.” White report § 108.

This is not correct. The compiled InfoPump Basic program that runs in the server is not a script; it is a p-code pseudo machine language program. The InfoPump interpreter processes low-level p-code statements, not script commands. “In the case of scripted requests, the interpreter scans the compiled script header..., and allocates the data space requested by the script. Then, the p-code statements are executed until the script ends or an unhandled error is encountered.” DEFS00025509.

The Court has defined “script” as “a group of commands to control data movement into and out of the system, and to control data transformation within the system.” Preliminary Claim Construction pp. 1-2. One skilled in the art would have recognized, as the Ellis-Doyle patent states, that a script processor “identifies the script command and invokes the correct method for that script command.” Ellis-Doyle patent 4:43-45. For the InfoPump interpreter to be a script processor, it would have to identify script commands and invoke the correct method for each script command.

To the extent that there are script commands in an InfoPump Basic program, the identification of those “commands” and determination of a possible correct method to invoke is not performed by the interpreter, but by the compiler, which is not a part of the interpreter. The interpreter processes only low-level p-code statements. DEFS00025513-DEFS00025514.

For example, Mr. White contends that InfoPump scripts “include commands to control data transformation with the system through the invocation of rules embedded in the script, or incorporated into a code fragment.” White report § 106. I disagree. An embedded set of rules does not form a script command. “The script is preprocessed, which involves converting everything except string literals to lower case ... In addition,

any code fragments are incorporated into the actual script text during this phase.”

DEFS00025511. Code fragments are processed by the compiler, not by the interpreter. The output of the compiler is a compiled p-code program consisting of a series of low-level statements in the InfoPump p-code instruction set.

The InfoPump p-code instruction set consists of “six statement types, each represented by a single ASCII character and followed by statement parameters.” The six statement types are Assignment (A), Handler call (H), Function call (F), Transfer (T), Miscellaneous (M), and Error checkpoint (E). DEFS00025513-DEFS00025514. None of these statement types constitutes a command to control data transformation.

To the extent that there is a command that controls data transformation, such a command would be represented by a series of low-level p-code statements generated by the compiler for the command. The interpreter does not process the command as such but simply processes individual p-code statements. The interpreter does not identify the command or determine which method to invoke for the command.

Thus InfoPump did not have the claimed script processor. It would not have been obvious to add a script processor to InfoPump because InfoPump utilizes a compiled p-code approach that uses a p-code virtual machine to processes low-level p-code instructions at runtime. InfoPump is designed for runtime efficiency, which is provided by the compiled p-code approach. The InfoPump scripting language “is implemented as a machine-independent p-code compiler for execution speed and portability to other environments.” DEFS00097225. For simple requests, InfoPump even bypassed p-code execution for maximum runtime efficiency. “In the case of simple requests, the p-code statements are bypassed since the instruction set is always the same. A function called

RunQuery is streamlined to handle simple requests with maximum efficiency, bypassing unnecessary interpreter elements.” DEFS00025509. Given the concern for runtime efficiency, it would not have been obvious to add a script processor because a higher-level script processor would have lower runtime efficiency, contrary to InfoPump’s design goals. Furthermore, adding scripts to InfoPump would have required a significant redesign and programming effort.

4. InfoPump did not include *1d) a rule set processor responsive to said script processor for manipulating a data bag for storing imported data and a data bag for storing export data*

Rule set processor

There is no identifiable component in InfoPump that could be considered to be a rule set processor. The same interpreter that executes the p-code for compiled scripted requests executes the p-code for the transformations within those requests. The p-code for the transformations is not executed responsive to a script processor; the p-code for the transformations is already in the compiled scripted request. It is executed when the interpreter encounters it.

Mr. White apparently contends that the rule set processor is the portion of the interpreter that implements assignment, transfer, and function p-code statements. “This software component includes the assignment, transfer, and function modules, which are called by the interpreter when it encounters an assignment, transfer, or function statement identifier while executing a script.” White § 109. For support, he cites the ISL p-code (Instructions) [DEFS00025513-DEFS00025514] section of the InfoPump Technical Document.

However, the assignment, transfer, and function statement types are three of the

six p-code statement types. “There are six statement types, each represented by a single ASCII character and followed by statement parameters.” DEFS00025513.

In fact, the assignment, transfer, and function statement types represent the three p-code statements responsible for general processing, where general processing refers to processing other than input/output (I/O) or error-handling: Assignment (A), Function call (F), and Transfer (T). The remaining three statement types do not handle general processing: Handler call (H) (responsible for input/output), Miscellaneous (M) (miscellaneous events, including error handling), and Error checkpoint (E) (error handling).¹

¹ There are also two other statement types representing line numbers for error messages (L) and the end of the script (X), but those do not perform processing actions.

- Assignment (A) - assign the result of expression <expression> to value <result-value>
A <result-value> <expression>
- Handler call (H) - invoke communications handler short <handler-enum> using profile handle along <profile-handle> with ulong <parm-count> parameters, and assign the result to <result-value>. The parameters are given in a list, each indicating the parameter name by keycode in ushort <key-enum> with the value in <key-value>
H <result-value> <handler-enum> <profile-handle> <parm-count> [<key-enum> <key-value>...]
WHERE <handler-enum> is one of:
COM_CONNECT
COM_DISCONN
COM_SEND
COM_RECEIVE (currently unused)
COM_LOAD
COM_STORE
COM_QUERY (currently unused)
COM_CHANGE (currently unused)
- Function call (F) - invoke library function short <function-enum> with ulong <parm-count> parameters, and assign the result to <result-value>. The parameter list is simply a list of values
F <result-value> <function-enum> <parm-count> [<value>...]
WHERE <function-enum> is a member of the C enumeration library_functions (LIBFUNC_XXX)
- Transfer (T) - transfer of control type short <transfer-enum> to the label at script data offset along <offset>. In the case of a conditional transfer (<misc-enum> = CONDITIONAL_JUMP), evaluate the boolean expression <conditional-expression>, and only transfer control if the result is FALSE.
T <transfer-enum> <offset> [<conditional-expression>]
WHERE <transfer-enum> is one of:
UNCONDITIONAL_JUMP
CONDITIONAL_JUMP
UNCONDITIONAL_CALL
UNCONDITIONAL_RETURN
- Miscellaneous (M) - a miscellaneous event has occurred, indicated by short <misc-enum>
M <misc-enum> variable-parameters
WHERE the values of <misc-enum> and their parameters are:
 - GEN_ONERR <transfer-enum> <label-offset>
To define an ON ERROR error handler
WHERE short <transfer-enum> is either UNCONDITIONAL_JUMP or CONDITIONAL_JUMP, and ulong <label-offset> is the offset in script data of the label's value
 - GEN_ONPROVERR <transfer-enum> <label-offset>
To define an ON COMMERROR communications error handler
WHERE short <transfer-enum> is either UNCONDITIONAL_JUMP or CONDITIONAL_JUMP, and ulong <label-offset> is the offset in script data of the label's value
 - GEN_TRANSACTION <transact-enum>
To define a transaction operation
WHERE <transact-enum> is one of {BEGIN_XACT, COMMIT_XACT, ROLLBACK_XACT}
 - GEN_PROVIDER <profile-handle> <ldb-profile-name-value> <profile-handle-string>
To define a profile declaration
WHERE ulong <profile-handle> is the profile's handle, <ldb-profile-name-value> is the value object of string constant type containing the name of the ldb profile object, and <profile-handle-string> is the name of the profile handle as defined in the script
 - GEN_FRAGMENT <fragment-name-string>
To indicate that the following code is in the given code fragment (for error messages)
WHERE <fragment-name-string> is the name of the ldb code fragment
- Error checkpoint (E) - check if any errors have occurred, and if so, handle them at this point (occurs every time temporary variables are freed up).
E
- Line Number (L) - set the line number to ulong <line-number> (for error messages)
L <line-number>
- End of Script (X) - no more pcode
X

DEFS00025513-DEFS00025514.

Far from implementing a separate rule set processor responsive to the InfoPump interpreter, as Mr. White contends, the assignment, transfer, and function modules in fact

Rebuttal Expert Report of
Walter G. Rudd

are core parts of the InfoPump interpreter and are not separable from it. The interpreter cannot process non-trivial InfoPump programs without the assignment, transfer, and function modules.

The use of the assignment, transfer, and function modules in processing an entire InfoPump program (not just portions implementing rule-set functionality) can be seen in a sample InfoPump program in the InfoPump Script Language Guide. DEFS00107888-DEFS00107892, CA00177477-CA00177481.

This particular sample program can be roughly characterized as having six sections.

First, there is a section declaring data elements such as variables and user-defined structures. DEFS00107888-DEFS00107889. That section is processed by the compiler to generate parts of the compiled script header used to “define the data space and format of various script data elements” and does not directly produce p-code statements. DEFS00025511.

```
REM ***** Begin with declarations

DEF buffer, line, field, table AS STRING
DEF filesize, fileptr, rows, index, loc, size AS INT

TYPE sales_def AS STRUCTURE
(
  Region VARSTRING (10),
  Q1_Sales INT,
  Q2_Sales INT,
  Q3_Sales INT,
  Q4_Sales INT
)
DEF sales AS sales_def
```

DEFS00107888-DEFS00107889; CA00177477-CA00177478.

Second, there is a section performing initialization and setup. This section

includes an assignment statement (“DEF pro as PROFILE = “SQLServerProfile””) and a function call (“MESSAGEOUTPUT (TRUE, NULL, FALSE)”) that, when compiled, would generate Assignment (A) and Function call (F) p-code statements respectively.

```
DEF pro AS PROFILE = "SQLServerProfile"

REM ***** Script Initialization and Setup

MESSAGEOUTPUT (TRUE, NULL, FALSE)
ON COMMERROR GOTO commerr_label

CONNECT pro ( )
```

DEFS00107889; CA00177478.

Third, there is a section that reads a buffer of data without transformation. This section includes numerous assignment statements (e.g., “index = 1”), function calls (e.g. “MESSAGE(...”), and transfer statements (e.g., “IF ... THEN ... ELSE ... END IF”).

```
REM ***** Read in from the file 20000 characters at a time

filesize = FILESIZE ("sales.dat")
IF (filesize > 20000)
  THEN size = 20000
  ELSE size = filesize
  END IF
filesize = filesize - size
fileptr = size
IF NOT FILEREAD ("sales.dat", buffer, 1, size)
  THEN MESSAGE ("cannot read file")
  STOP
end if
```

DEFS00107889; CA00177478.

Fourth, there is a section that gets the table name and clears the corresponding table. This section includes assignment statements (e.g., “index = 1”) and function calls (e.g, “MESSAGE (...”).

```

REM **** Get the first row (table name), and clear the SQL Server table

index = 1
loc = INSTR (index, buffer, "ln")
table = MID (buffer, index, loc - 2)
index = loc + 1

SEND pro (#query "delete from " + table)

MESSAGE ("Beginning transfer at ", DATEGET ( ))

```

DEFS00107890; CA00177479.

Fifth, there is a loop that processes rows, one at a time. DEFS00107890-DEFS00107892 The loop can be further characterized as having four subsections.

The first subsection sets up the loop. This subsection includes an assignment statement ("rows = 0") and a transfer statement ("DO WHILE (TRUE)").

```

REM **** Scan in rows, one at a time, until all have been read

rows = 0
DO WHILE (TRUE)

```

DEFS00107890; CA00177479.

The second subsection reads a buffer of data without transformation. This subsection includes numerous assignment statements (e.g., "index = 0"), function calls (e.g., "MESSAGE (...)", and transfer statements (e.g., "IF ... THEN ... ELSE ... END IF").

```

REM **** When there are less than 1000 characters left in the buffer,
REM **** read in another 20000

IF (index > 19000 and filesize > 0)
  THEN IF (filesize > index)
    THEN size = index - 1
    ELSE size = filesize
    END IF
  IF NOT FILEREAD ("sales.dat", field, fileptr + 1, size)
    THEN MESSAGE ("cannot read file")
    STOP
  END IF
  fileptr = fileptr + size
  filesize = filesize - size
  buffer = RIGHT (buffer, 20000 - index + 1) + field
  index = 0
END IF

```

DEFS00107890; CA00177479.

The third sub-section transforms data from the buffer and stores the transformed data into fields in the “sales” user-defined structure. If any part of the program contains rule sets, this subsection would be the one.

```

REM ***** Scan in the five fields, using the delimiters to determine
REM ***** where each field ends. For the four number fields, use the
REM ***** convert function. The delimiter is comma, except for the
REM ***** last field, where it is a newline.

```

```

REM ***** When scanning the first field, there may not be a comma if
REM ***** the previous line was the final row.

```

```

    loc = INSTR (index + 1, buffer, ",")
    IF (loc = 0)
        THEN BREAK
    END IF
    sales.Region = MID (buffer, index, loc - index)
    index = loc + 1

    loc = INSTR (index + 1, buffer, ",")
    CONVERT (MID (buffer, index, loc - index), sales.Q1_Sales)
    index = loc + 1

    loc = INSTR (index + 1, buffer, ",")
    CONVERT (MID (buffer, index, loc - index), sales.Q2_Sales)
    index = loc + 1

    loc = INSTR (index + 1, buffer, ",")
    CONVERT (MID (buffer, index, loc - index), sales.Q3_Sales)
    index = loc + 1

```

```

REM ***** When scanning the last field, there may not be a newline if
REM ***** it is the final field. If so, set the current location to
REM ***** the file size, which will insert the row, but will cause
REM ***** the loop to fail the next time through.

```

```

    loc = INSTR (index + 1, buffer, "\n")
    IF (loc = 0)
        THEN loc = filesize
    END IF
    CONVERT (MID (buffer, index, loc - index), sales.Q4_Sales)
    index = loc + 1

```

DEFS00107891-DEFS00107892; CA00177480-CA00177481.

The fourth subsection stores the transformed data into the SQL server and transfers control back to the beginning of the loop. This subsection includes an assignment statement (“rows = rows + 1”) and a transfer statement (“LOOP”).

```

REM ***** The row has been scanned in. Insert it into the SQL Server
REM ***** table. Use bulk transfer in order to speed up the operation.

STORE pro (#data sales, #table table, #create TRUE,
          #bulk TRUE)

rows = rows + 1
LOOP

```

DEFS00107892; CA00177481.

After the loop, the sixth and final section of the program disconnects and completes the script. This section includes function calls (such as “MESSAGE(...)”) and transfer statements (such as “STOP”).

```

REM ***** The file transfer to SQL Server is complete. Disconnect and
REM ***** complete the script.

MESSAGE ("Transfer of ", rows, " complete at ",
DATEGET ())

DISCONNECT pro ()
STOP

REM ***** Error handler
LABEL commerr_label

MESSAGE (COMMERRFORMAT ())
DISCONNECT pro ()
STOP

```

DEFS00107892; CA00177481.

As seen above in the sample InfoPump program, assignment, function call, and transfer statements occur throughout an InfoPump program, and not only in a section that transforms data using rule sets. Likewise, the assignment, function call, and transfer modules are used during processing of the entire program, and not only in a section with rule-set statements.

The assignment, function call, and transfer modules are therefore integral parts of the InfoPump interpreter. They are not part of a separate processor that processes only rule sets. Contrary to Mr. White's contention, they are not part of a rule set processor responsive to the script processor.

I have seen no evidence that InfoPump included a rule set processor responsive to the script processor. In my opinion, InfoPump did not include a rule set processor responsive to a script processor. As seen above, InfoPump did not have a script processor. Furthermore, the InfoPump interpreter was a single processor that processed all p-code statements. InfoPump did not have a separate rule-set processor for processing data transformation rule sets.

It would not have been obvious to add a rule set processor responsive to a script processor. First, as is noted above, it would not have been obvious to add a script processor to InfoPump. Second, InfoPump provided a single programming language based on BASIC that allowed users to design data transformation programs using a simple and familiar programming language. "Scripts are written in the InfoPump Script Language. It is based on the simplest programming language, BASIC, so it will be familiar to many users and easy to learn." CA00177372, DEFS00107783. Adding a second processing language and processor for data transformation rule sets would have added complexity without obvious benefit. In terms of implementation, adding a second processor to InfoPump's p-code compiler and interpreter would require additional development cost and complication without obvious benefit. Having InfoPump users learn a new, second language for data transformation rule sets would be contrary to

InfoPump's goal of providing users with a simple and familiar BASIC-like programming language.

Data bags

InfoPump did not include data bags. Mr. White claims that user-defined structures are data bags and that "User-defined structures are stored in non-persistent memory, are created by the software component processing the InfoPump scripts [i.e., the interpreter, see White report § 108]..." White report § 110. The interpreter identified by Mr. White as the script processor allocates all the space the program needs and then interprets the p-code. "In the case of scripted requests, the interpreter scans the compiled script header (see compiler description under InfoPump Manager), and allocates the data space required by the script. Then, the p-code statements are executed until the script ends or an unhandled error is encountered." DEFS00025509. There is no indication here that the interpreter, or any component of it, creates user-defined structures. Like the code that performs any other operations in a compiled scripted request, the code that creates user-defined structures is embedded in the p-code.

It would not have been obvious to add data bags created by a script processor because as seen above, InfoPump did not have a script processor and it would not have been obvious to add a script processor to InfoPump.

Documents

Mr. White states that the InfoPump 1.0 User Guides (The InfoPump Installation and Server Guide, The InfoPump Manager Guide and The InfoPump Script Language Guide) were "distributed in a single volume" and "thus themselves collectively constitute a single prior-art printed publication" and that the Guides "as a prior-art publication

anticipate and render obvious” the Doyle-Ellis Patent claims. White report § 86. Mr. White does not explain what “distributed in a single volume” means or what is the specific evidence supporting that conclusion. Contrary to his conclusion, the contents of the Guides show that each is a separate publication. For example, each Guide contains a cover page, a copyright notice, a table of contents, an index and what appears to be, to the right of each copyright notice, a publication designation number. Neither the Guides individually nor the Guides collectively, or the document “InfoPump Commonly Asked Questions” (DEFS00025759-79) discloses scripts, a metadata database, a script processor, a rule set processor, or data bags.

Claim 12

While Mr. White asserts that claims 1-11 and 13-19 are anticipated by InfoPump, Mr. White does not make such an assertion as to claim 12 and, instead, asserts that claim 12 is obvious in light of InfoPump in combination with other cited references. Thus, Mr. White’s report implies that claim 12 is *not* anticipated by InfoPump. I agree that claim 12 is not anticipated by InfoPump.

Claim 12 is not obvious in light of InfoPump and the cited references. Claim 12 is dependent on claims 11, 10 and 1 and the elements of those claims are not disclosed by InfoPump alone or in combination with the cited references. Further, Mr. White asserts that there would have been motivation to combine the discussed commands with InfoPump “because they were all used in connection with handling data from heterogeneous data sources and data targets.” White Report § 137. I understand that motivation to combine references as Mr. White suggests requires more than a vague

notion that the references are in the same field. One skilled in the art would not have recognized such a combination as obvious.

In conclusion, InfoPump lacks many of the elements of independent claims 1 and 17 of the Ellis-Doyle patent and therefore does not anticipate either of those claims. The remainder of the asserted claims are dependent claims from claim 1 or claim 17, and hence none of the asserted claims is anticipated by InfoPump. InfoPump, alone or in combination with any and/or all of the references cited by White for combination with InfoPump, does not render obvious any of the asserted claims.

B. The Coleman Patent

Mr. White claims that U.S. Pat. No. 5,708,828 (the “Coleman patent”) “anticipates and/or renders obvious each of claims 1-19 of the ‘662 patent.” White report § 155. I disagree. The Coleman patent neither anticipates nor renders obvious any of the asserted claims. It does not include any of the elements of any of the asserted claims. The Coleman Patent does not include the claimed distribution system, scripts, metadata database, script processor or data bags.

1. The Coleman patent does not disclose *a distribution system for transforming and exchanging data between heterogeneous computer systems*,

The Coleman patent describes converting data from one flat data file into another. These flat files must be created and loaded by computer systems that are not included in the Coleman device. This means that the Coleman system does not import data from or export data to computer systems not included in the Coleman device. The Coleman patent describes a data conversion system. The court construed “distribution system” as “a computer system for importing data from a source computer system, transforming the

Respectfully submitted,

A handwritten signature in black ink, appearing to read "Walter G. Rudd". The signature is fluid and cursive, with the first name "Walter" being more legible than the last name "Rudd".

Walter G. Rudd June 16, 2009